Alcom-FT Technical Report Series
ALCOMFT-TR-03-27

Fast Bound Consistency for the Global Cardinality Constraint

Irit Katriel* and Sven Thiel**

Max-Planck-Institut für Informatik, Saarbrücken, Germany,
{irit,sthiel}@mpi-sb.mpg.de

Abstract. We show an algorithm for bound consistency of *global cardinality constraints*, which runs in time $O(n+n')$ plus the time required to sort the assignment variables by range endpoints, where n is the number of assignment variables and n' is the number of values in the union of their ranges. We thus offer a fast alternative to Régin's arc consistency algorithm [6] which runs in time $O(n^{3/2}n')$ and space $O(n \cdot n')$. Our algorithm can also narrow the bounds for the number of occurrences of each value, which has not been done before.

1 Introduction

The *Global Cardinality Constraint* $GCC(x_1, \cdots, x_n, c_{v_1}, \cdots, c_{v_{n'}})$ is specified on n assignment variables $x_1, \ldots, x_n$ and n' count variables $c_{v_1}, \ldots, c_{v_{n'}}$. The idea is that each assignment variable x_j takes a value in $D = \{v_1, \cdots, v_{n'}\}$ and each value v_i is used exactly c_{v_i} times. With each assignment variable x_j we associate a domain $D_j \subseteq D$, and the domain of a count variable c_{v_i} is an interval $E_i = [L_i, U_i]$. For a tuple $t \in D^n$ and $v \in D$ denote by $occ(v,t)$ the number of occurrences of the value v in t. Then the set S containing all solutions of the constraint is defined as follows:

$$S = \{(w_1, \ldots, w_n; o_1, \ldots, o_{n'}) \mid \forall_j w_j \in D_j \ \wedge \ \forall_i occ(v_i, (w_1, \ldots, w_n)) = o_i \in E_i\}$$

Given a constraint with ranges for the variables, the first question is whether $S \neq \emptyset$, which means that there is an assignment of values to the variables which satisfies the constraint. The *arc consistency* problem for the assignment variables is to reduce the domains of these variables such that D_j is the projection of S onto its jth component. In the *bound consistency* problem we assume $v_1 < \ldots < v_{n'}$ and each D_j is a contiguous interval of values, i.e. $D_j = [\underline{D}_j, \overline{D}_j]$. The problem is to shrink the intervals of both the assignment and the count variables to the minimum sizes such that $S \subseteq D_1 \times \cdots \times D_n \times E_1 \times \cdots \times E_{n'}$. I.e., the domain of the kth variable is bound consistent iff S contains at least one tuple whose kth component equals the smallest (largest) value in the respective domain.

Régin [6] has provided an algorithm that achieves arc consistency for the assignment variables of a GCC, but does not reduce the domains of the count

* Partially supported by DFG grant SA 933/1-1.
** Partially supported by the EU IST Programme, IST-1999-14186 (ALCOM-FT).

variables. He constructs a bipartite graph and computes a maximum flow in it. He then defines the residual graph with respect to this flow, finds the strongly connected components (SCCs) in the residual graph and discards the edges between SCCs that carry no flow. The algorithm runs in time $O(n^{3/2}n')$ and space $O(n \cdot n')$ and is dominated by the complexity of finding a maximum flow using Ford and Fulkerson's algorithm [3].

Mehlhorn and Thiel [5] considered the AllDifferent constraint, which requires that each variable is assigned a different value and is the special case of GCC in which $[L_i, U_i] = [0, 1]$ for all $1 \leq i \leq n'$. They show an algorithm for bound consistency of this restricted case. Though phrased in terms of bipartite matchings, it can also be interpreted as a simplified version of Régin's flow-based algorithm; the maximum flow in this case is a maximum matching in the graph. Their algorithm runs in linear time plus the time required to sort the variables according to the endpoints of their ranges. The improved running time compared to Régin's algorithm is achieved due to the fact that the algorithm exploits the simpler structure of the graph: The assumption that the range of each variable is an interval implies that the bipartite graph is *convex*, which means that the neighborhood of each variable node is a contiguous sequence of value nodes.

For GCC, we show an algorithm of the same flavor that achieves bound consistency for the assignment variables and can also narrow the bounds of the count variables (but in general not to bound consistency). A significant difference between the problems is that in an AllDifferent constraint, only variable nodes must be matched but a value node may or may not be matched. Now, we also have lower bound requirements on the value nodes. We will show, again using matching terminology, that it is still possible to find a maximum flow in the bipartite graph and compute the SCCs in the residual graph in time which is linear in the number of nodes plus the time required for sorting the range endpoints of the assignment variables. This algorithm can be used as a fast and space efficient alternative to Régin's algorithm where bound consistency is enough, or as a preprocessing step to speed up Régin's algorithm where arc consistency is desired. For the full version of this paper, refer to [4]. Source code is available by request from the authors.

2 Preliminaries

2.1 Normalization of the x-ranges

The x-ranges are called *normalized* if $D = \{1, \cdots, n'\}$. This has the advantage that we can use the values as array indices. Normalization can be achieved by identifying each value v_i with its index i in the sorted order $v_1 \leq \cdots \leq v_n$. To do this we need to compute for every original domain $D_j = [v_{l_j}, v_{h_j}]$ the corresponding normalized domain $D'_j = [l_j, h_j]$. After sorting the endpoints of $D_1, \cdots, D_n$ in ascending order, this can be done in time $O(n + n')$. If the v_i's are integers drawn from the range $[1, n^k]$ for some fixed k, sorting can be done in time $O(n)$. After achieving bound consistency for the normalized domains $D'_1, \ldots, D'_n$, we can easily narrow the original domains to bound consistency.

2.2 A generalization of matching

A *matching* in a graph is a subset M of its edges such that each node is adjacent to at most one edge in M. We generalize this notion to capacitated graphs $G = (V, E, C)$ where C is a function that maps every node $\nu \in V$ to an interval $C(\nu) = [L_\nu, U_\nu]$. We call $C(\nu)$ the *capacity requirement* of ν. For a set M of edges and a node ν we denote by $M(\nu)$ the set of all nodes that are adjacent to ν by an edge in M. A *generalized matching* in G is a subset M of its edges such that for each node $\nu \in V$ we have $|M(\nu)| \in [L_\nu, U_\nu]$. We call $M(\nu)$ the set of *matching mates of* ν.

As in [5], we define the *intersection graph* of a GCC, which is in this case a capacitated bipartite graph with n nodes $\{x_1, \cdots, x_n\}$ representing the variables on one side and n' nodes $\{y_1, \cdots, y_{n'}\}$ representing the values in the ranges of the variables on the other side. The capacity ranges of the variable nodes are all $[1, 1]$, indicating that each variable must be assigned exactly one value. For a value node, the capacities are according to the count requirements of the value represented by the node: $\forall_{1 \le i \le n'} [L_{y_i}, U_{y_i}] = [L_i, U_i]$, and $\forall_{1 \le j \le n} [L_{x_j}, U_{x_j}] = [1, 1]$. The edge (x_j, y_i) exists in the graph if and only if $i \in [\underline{D_j}, \overline{D_j}]$.

Running example: Throughout the paper, we will illustrate the algorithms with the following example (see Figure 1): $GCC(x_1, \cdots, x_6, c_{v_1}, \cdots, c_{v_4})$ where the assignment variable ranges D_j and the count variable ranges E_i are

E_1	E_2	E_3	E_4
[1,3]	[1,2]	[1,1]	[1,1]

D_1	D_2	D_3	D_4	D_5	D_6
[1,1]	[1,2]	[1,2]	[2,2]	[3,4]	[3,4]

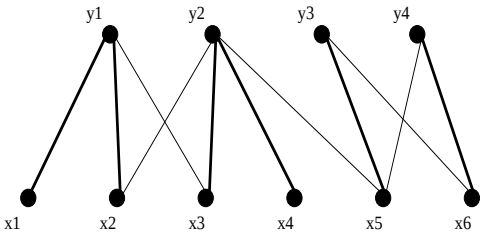

Fig. 1. The intersection graph of the example GCC and a generalized matching in it, marked with bold edges.

Lemma 1. *Let T denote the projection of S onto its first n components. Every generalized matching $M = \{\{x_i, y_{g(i)}\} | 1 \le i \le n\}$ corresponds to the tuple $(g(1), \cdots, g(n))$ in T and vice versa.*

Proof. Immediate from the definition of a generalized matching. □

2.3 The connection to Régin's algorithm

Régin's algorithm computes a maximum flow in a graph $\vec{H}$ which is similar to our intersection graph with a few differences: It is directed, contains two

additional nodes s and t, and the capacity requirements are on the edges and not on the nodes. The edge set of $\vec{H}$ is defined as follows.[1] There is an edge (x_j, y_i) with capacity requirement $[0, 1]$ iff $i \in D_j$. For every y_i there is an edge (y_i, s) with capacity bounds $[L_i, U_i]$, and for every x_j we have the edge (t, x_j) with capacity requirement $[0, 1]$. Régin shows that the constraint has a solution iff the maximum flow from t to s has value n.

His algorithm searches for a flow F of value n. If none exists, it reports failure. Otherwise, it constructs the residual graph $\vec{R}_F$, which allows to determine for each variable the arc consistent values in its domain. $\vec{R}_F$ has the same nodes as $\vec{H}$ (except for t) and the following edges: An edge (x_j, y_i) iff $i \in D_j \wedge F(x_j, y_i) = 0$, an edge (y_i, x_j) iff $i \in D_j \wedge F(x_j, y_i) = 1$, an edge (y_i, s) iff $F(y_i, s) < U_i$, and finally an edge (s, y_i) iff $F(y_i, s) > L_i$. In Corollary 2 in [6], Régin proves that a value i in the domain D_j is consistent for x_j iff $F(x_j, y_i) = 1$ or x_j and y_i belong to the same SCC of $\vec{R}_F$.

A generalized matching M in the intersection graph G corresponds to the flow F_M in $\vec{H}$ such that for each edge (x_j, y_i), $F_M(x_j, y_i) = 1$ if $\{x_j, y_i\} \in M$ and $F_M(x_j, y_i) = 0$ otherwise; $F_M(y_i, s) = |M(y_i)|$ for all i and $F_M(t, x_j) = 1$ for all j. Observe that the total flow from t to s has value n.

As in [5], we augment the residual graph $\vec{R}_{F_M}$ such that we obtain a graph $\vec{G}$ with the following property: A value $i \in D_j$ is arc consistent iff x_j and y_i are in the same SCC of $\vec{G}$. If x_j is matched with y_i in M, $\vec{R}_{F_M}$ contains the edge (y_i, x_j), and in $\vec{G}$ we add the edge (x_j, y_i) in the opposite direction. So if x_j and y_i are matched (equivalently, $F_M(x_j, y_i) = 1$) then they belong to the same SCC of $\vec{G}$. We call $\vec{G}$ the *oriented intersection graph*. Lemma 2 follows easily from the discussion above and Régin's results cited in it:

Lemma 2. *An edge $\{x_j, y_i\}$ in G belongs to a generalized matching iff x_j and y_i are in the same SCC of $\vec{G}$.*

This lemma implies the correctness of the following bound consistency algorithm for the assignment variables:

1. Find a generalized matching in G.
2. Construct $\vec{G}$ and compute its SCCs.
3. Narrow the ranges as much as possible such that they still represent all edges within the SCCs.

We want to point out that our algorithm uses $O(n + n')$ space and does not construct any of these graphs explicitely.

3 Finding a generalized matching in the intersection graph

The algorithm for finding a generalized matching in a convex graph is given in Figure 2. It makes three passes over the y nodes. In the first two passes it goes

[1] The edges in $\vec{H}$ have the opposite direction compared to Régin's graph. We prefer the reversed orientation because it is equivalent and simplifies the following presentation.

from left to right and uses a priority queue P to which x nodes are inserted when they become candidates for matching and in which they are sorted according to the upper endpoints of their domains. That is, the node x that is extracted from P by an *ExtractMin* operation is the one whose domain ends earliest. So any node that remains in P can match the same future y-nodes as x (by convexity), but maybe even more. And hence, it is reasonable to extract x and keep the others.

In the first pass the algorithm ignores the lower bound capacities and finds a generalized matching in the graph $G_{L=0} = (V, E, C_{L=0})$, where $C_{L=0}(y_i) = [0, U_i]$ for all i and $C_{L=0}(x_j) = [1, 1]$ for all j. $G_{L=0}$ is the same as G except that the lower bound capacities of all y nodes are zero. This generalized matching is constructed as follows: The y nodes are traversed from $y_1, \cdots, y_{n'}$. When y_i is reached, all of the x nodes that are connected to y_i, but not to any node with a smaller index, are inserted into the queue; they are now candidates for matching. Then y_i is matched with up to U_i nodes from P (less if P does not contain that many nodes). If, while processing y_i, the algorithm extracts a node from the queue which is not connected to y_i, it reports failure (cf. Lemma 3).

In the second pass, the algorithm makes another traversal of the y nodes in the same order, but this time it ignores the upper bounds and constructs a generalized matching in the graph $G^{U=\infty} = (V, E, C^{U=\infty})$, where $C^{U=\infty}(y_i) = [L_i, \infty]$ for all i and $C^{U=\infty}(x_j) = [1, 1]$ for all j. In this matching, each y node is matched with the minimal number of x nodes such that its lower capacity bound is respected and all x nodes which are not connected to y nodes with higher indices are matched. If the queue becomes empty while the algorithm tries to fulfil the lower capacity bound L_i for node y_i, the algorithm reports failure (see Lemma 6).

The matching found in the first pass is used during the second pass to determine the order in which nodes are inserted into the queue. We will show that if a generalized matching exists, then there also exists a generalized matching such that each y_i is matched only with nodes that were matched in the first pass with nodes from $y_1, \cdots, y_i$ (cf. Lemma 4). Hence, when y_i is reached in the second pass, the nodes that were matched with it in the first pass are inserted into the queue as candidates for matching with it.

The matching found in the second pass may violate the upper capacity bounds in G. The third pass corrects this by traversing the y nodes from $y_{n'}$ to y_1 and shifting these excesses to y nodes with lower indices.

The table below shows the mate of x_j in our running example after each pass of the algorithm.

j	1	2	3	4	5	6
pass 1	1	1	1	2	2	3
pass 2	1	2	2	2	3	4
pass 3	1	1	2	2	3	4

```
(* Input: (1) Ranges of assignment variables D_j = [D_j, D̄_j] for each x_j, 1 ≤ j ≤ n *)
(*        (2) A capacity requirement [L_i, U_i] for each 1 ≤ i ≤ n'. *)
(* Output: A generalized matching if it exists, failure otherwise. *)

(* 1st pass: find a matching in G_{L=0} (encoded in m) *)
P ← [] (* priority queue containing x nodes sorted according to D̄ *)
j ← 0; U_{n'+1} ← ∞
for i = 1 to n' + 1 do
    forall x_h with D_h = i do P.Insert x_h
    u ← 0
    while P is not empty and u < U_i do
        j ← j + 1; x_{f(j)} ← P.ExtractMin
        m[f(j)] ← i; u ← u + 1
        if D̄_{f(j)} < i then report failure
        (* No failure implies i ∈ D_{f(j)} *)
    end
    β_i ← j
endfor

(* 2nd pass: find a matching in G^{U=∞} and ensure feasibility for G *)
j ← 0; α_0 ← 0
for i = 1 to n' do
    forall x_h with m[h] = i do P.Insert x_h
    for ℓ = 1 to L_i do
        if P is empty then report failure
        j ← j + 1; x_{g(j)} ← P.ExtractMin
        match y_i ↔ x_{g(j)}   (* m[g(j)] ≤ i *)
    endfor
    while P is not empty and P.MinPriority < i + 1 do
        j ← j + 1; x_{g(j)} ← P.ExtractMin
        match y_i ↔ x_{g(j)}   (* m[g(j)] ≤ i *)
    end
    α_i ← j
endfor

(* 3rd pass: transform the matching from 2nd pass into a matching in G *)
for i = n' to 1 do
    if y_i has currently h mates s.th. e := h − U_i > 0 then
        for k = 1 to e do
            choose a current mate x_{j_k} of y_i with i' := m[j_k] < i
            match y_{i'} ↔ x_{j_k}   (* y_i looses x_{j_k} *)
        endfor
    endif
endfor
```

Fig. 2. Algorithm for generalized matching in a convex capacitated bipartite graph

Lemma 3. *If the algorithm reports failure in the first pass, then there is no generalized matching in $G_{L=0}$. And hence, there is also none in G.*

Proof. Suppose the algorithm reports failure in iteration i after extracting a node x_j (observe that any extraction in iteration $n'+1$ causes failure.) Then in iteration $i-1$ it only extracts x nodes that are not connected to y_i, and P contains x_j afterwards. Let i' be the maximum iteration before $i-1$ such that an x node connected to y_i is extracted or P becomes empty during iteration i'. If no such iteration exists, choose $i'=0$. Let X denote the x nodes extracted in iterations $i'+1,\cdots,i-1$. Since P never becomes empty during these iterations, we have $|X| = \sum_{k=i'+1}^{i-1} U_k$, i.e. X exhausts the capacities of the nodes in $Y = \{y_{i'+1},\cdots,y_{i-1}\}$.

The nodes in X are not connected to y_i or a y node to its right. We show that there is also no connection to $y_{i'}$ or a y node to its left. Suppose otherwise, i.e. $x_\ell \in X$ is connected to $y_{i''}$ with $i'' \le i'$. As x_ℓ was inserted to P in iteration i' or earlier and is extracted in a later iteration, P was never empty in iteration i'. By the choice of i', this implies that a node $x_{j'}$ connected to y_i was extracted in iteration i'. As x_l was removed later, it must also be connected to y_i, a contradiction to the choice of i'. Thus all neighbors of the nodes in X are in Y. A similar argument proves that the node x_j, which caused the failure in iteration i, is only connected to nodes in Y.

Since a generalized matching in $G^{U=\infty}$ would have to match the nodes in Y with at least $|X|+1$ nodes, but can only match them with $|X|$ different nodes, such a matching cannot exist. □

For any matching M and subset $Y \subseteq \{y_1,\cdots,y_{n'}\}$, let $M(Y) = \bigcup_{y\in Y} M(y)$, i.e. $M(Y)$ is the set of all x nodes that are matched with nodes of Y.

Lemma 4. *If there is a generalized matching M' in G, then there is also a generalized matching M with the following property: For all edges $\{x_j, y_i\} \in M$, we have $m[j] \le i$.*
M can be chosen such that $|M(y_i)| = |M'(y_i)|$ for $i = 1,\cdots,n'$. And hence, it is impossible to match $\{y_1,\cdots,y_i\}$ with more than β_i nodes[2]*.*

Proof. Let M' be a generalized matching in G, and suppose it does not have the desired property. Let i be minimal such that there is an edge $\{x_j, y_i\}$ with $k = m[j] > i$. Not all nodes x_ℓ with $m[\ell] = i$ can be matched with y_i in M', otherwise neither the queue nor the capacity of y_i would have been exhausted after iteration i in the first pass. So there is an edge $\{x_{j'}, y_{i'}\} \in M'$ with $i' \neq i$ and $m[j'] = i$. By the choice of i we have $i' > i$. When $x_{j'}$ was extracted in the first pass, x_j was also in the queue. So we conclude that x_j is also connected to $y_{i'}$. Thus we can swap the mates of x_j and $x_{j'}$ and obtain a new generalized matching M''. M'' does not violate the property for the nodes $y_1,\cdots,y_{i-1}$ and the number of violations at y_i is less than in M'. This shows that we can transform the matching until we eventually obtain a matching M with the desired property.
The last statement follows from the fact that $\beta_i = |\{x_j \mid m[j] \le i\}|$. □

[2] We will refer to the β_i values in Section 5.

Lemma 5. *If the algorithm does not report failure in iteration i of the second pass, then for any generalized matching M in G we have $|M(\{y_1, \cdots, y_i\})| \geq \alpha_i$.*

Proof. By induction on i. For $i = 0$ and $i = 1$ the claim is easy to verify. So let us assume that it holds for $i' = 1, \cdots, i-1$ and prove it for i. If the body of the while-loop is not executed, we have $\alpha_i = \alpha_{i-1} + L_i$, and applying the induction for $i' = i - 1$ immediately proves the claim. So suppose that nodes are extracted in the while-loop, which implies that no x that is extracted in iteration i is connected to y_{i+1}. Let i' be the maximum iteration before i such that a node connected to y_{i+1} is removed. If no such iteration exists, choose $i' = 0$. Let X denote the x nodes extracted in the iterations $i' + 1, \ldots, i$, and let $Y = \{y_{i'+1}, \cdots, y_i\}$. A similar argument as in the proof of Lemma 3 shows that the neighbors of any node $x \in X$ are contained in Y. Thus $|M(Y)| \geq |X| = \alpha_i - \alpha_{i'}$. By the induction hypothesis for i', $|M(\{y_1, \cdots, y_i\})| \geq \alpha_{i'} + |M(Y)| = \alpha_i$. □

Lemma 6. *If the algorithm reports failure in the second pass, then there is no generalized matching in G.*

Proof. Suppose that the algorithm reports failure in iteration i, although a generalized matching M exists. By Lemma 4, we can assume that for any edge $\{x_j, y_k\}$ in M we have $m[j] \leq k$. So $y_1, \cdots, y_i$ are matched only with nodes that have been inserted into P so far. From the previous lemma we can conclude that there must be at least $\alpha_{i-1} + L_i$ such nodes. But since the algorithm reports failure, this is not the case, a contradiction to the existance of M. □

Lemma 7. *If the algorithm does not report failure, it constructs a generalized matching in G.*

Proof. If the second pass succeeds, we have a generalized matching in $G^{U=\infty}$, because we fulfil the lower capacity bounds of every y node (cf. the for-loop), and we match every x node with a neighbor on the y side (see the while-loop). So the only problem is the upper capacity bounds of the y nodes. The third pass takes care of these. Observe that this pass sweeps over the y nodes from right to left and distributes the excess mates of a node y_i only to y nodes with lower indices, so that it cannot increase the number of mates of a y node after it was processed. Furthermore, if x_j is matched with y_i at some point in time then $m[j] \leq i$. Since we have equality for at most U_i nodes, we can always select e excessive mates to distribute, if necessary. Suppose a node x_j is removed from y_i and matched with $y_{i'}$ in the third pass, where $i' = m[j]$. From the first pass, it is easy to see that $i' \in D_j$, i.e., x_j and $y_{i'}$ are connected. □

Implementing the algorithm in linear time (for normalized domains)

Now we discuss some implementation details of the algorithm. First we show how to implement it in time $O(n' + n \log n)$, and then we refine this implementation to obtain $O(n' + n)$ time. In the first variant we use a binary heap of size n to implement the priority queue P. Then the operations *Insert* and *ExtractMin*

take time $O(\log n)$ and *MinPriority* runs in constant time. Before we run the algorithm we sort the x nodes according to their lower range endpoints, which takes time $O(n+n')$ because all domains are in $[1,n']$. This sorting allows us to determine efficiently the nodes that have to be inserted into P in each iteration of the first pass. Recall that the order by which the x nodes are extracted in the first pass determines the order by which they are inserted in the second pass. So the two passes can be implemented in time $O(n'+n\log n)$. The third pass takes linear time. To see this we notice that every x node changes its mate at most once. So we maintain for every y node two seperate lists for the mates that it received in the second and the third pass respectively. Thus we can make sure that we process every x node only once.

In order to shave off the logarithmic factor in the running time, we have to find a faster implementation of the priority queue. As in [5] we simulate the priority queue by creating an instance of the offline-min problem [1, Chapter 4.8], which can be solved in linear time using a special union-find data structure [2]. (The offline-min algorithm needs a sorting of the x's according to their upper interval endpoints, which can be computed in time $O(n+n')$.)

Except for failure detection, the algorithm does not use the mapping f in the first pass at all. So the whole sequence σ of *Insert* and *ExtractMin* operations can be computed without knowing the results of the extractions. Checking if P is empty can be done by counting the number of insertions and comparing it with the number of extractions (which is equal to j). Failure detection can be done after the mapping f has been computed.

We give a brief description of the offline-min algorithm. When it is applied to a sequence σ it determines for every insertion the corresponding extraction. Let $E_1,\cdots,E_n$ be the extract operations in the order in which they occur in σ. The node x_j with minimum priority is the output of the first extraction E_k that follows its insertion. After deleting E_k from σ we process the x with next higher priority, and so on. We implement this with a union-find structure that maintains a partition $\mathcal{P}$ of $E_1,\cdots,E_n$. Every set in $\mathcal{P}$ has the form $\{E_h,E_{h+1},\cdots,E_k\}$, where all extractions except for E_k have been deleted. Suppose we process x_j and let E_s be the first extraction in the original sequence after the insertion of x_j. Observe that E_s may have been deleted already. To determine the extraction E_k for x_j, we simply have to find the set S in $\mathcal{P}$ containing E_s. Deleting E_k amounts to uniting S with the set of E_{k+1}.

We cannot use the offline-min algorithm directly for the second pass, because we need to know the result of the *MinPriority* operation in the while-loop online. But a slight enhancement will do the job (see Figure 3). We initialize our data structures as above but we only create a set for the mandatory extractions that are made in the for-loop. When we find a tentative extraction E_k for x_j, we verify that E_k occurs in iteration $i \le \overline{D}_j$. If not, then we know that x_j is extracted by the while-loop in iteration $\overline{D}_j$, and we do not delete E_k. We want to point out that we detect failure if there is a mandatory extraction for which no corresponding insertion is found. Assuming w.l.o.g. that $\sum_{i=1}^{n'} L_i \le n$, the algorithm runs in time $O(n'+n)$.

$k \leftarrow 1$ (* E_k will always be the next mandatory extraction *)
for $i = 1$ to n' **do**
    forall x_j with $m[j] = i$ **do** $s[j] \leftarrow k$ (* x's inserted in iteration i (before E_k) *)
    for $\ell = 1$ to L_i **do**
        create set $\{E_k\}$ labelled $\mathcal{L}_k^{(i,\ell)}$ (* for mandatory extraction ℓ in iter. i *)
        $k \leftarrow k + 1$
    endfor
endfor
create set $\{E_k\}$ labelled with $\mathcal{L}_k^{(n'+1,0)}$ (* dummy extraction *)
forall x_j sorted in ascending according to $\overline{D}$ **do**
    $S \leftarrow \text{find}(E_{s[j]})$; let $\mathcal{L}_k^{(i,\ell)}$ be its label
    if $i \leq \overline{D}_j$ **then**
        x_j is removed by extraction ℓ in the for-loop of iteration i
        unite S with the set S' containing E_{k+1} and
            label the union with the former label of S'
    else
        x_j is extracted by the while-loop in iteration $\overline{D}_j$
    endif
endfor

Fig. 3. Enhanced offline-min algorithm for the second pass

4 Finding the SCCs of the oriented intersection graph

As described in Section 1, we construct an oriented graph $\vec{G}$ from the intersection graph and the generalized matching M (see Figure 4). The edges which are not within a strongly connected component of $\vec{G}$ describe inconsistent assignments of values to variables. Mehlhorn and Thiel [5] gave an algorithm that finds the SCCs of a simpler oriented intersection graph in $O(n+n')$ time. Their graph does not contain the additional node s, which violates the convexity property of the graph because there is an edge from s to each node in $\{y_i : |M(y_i)| > L_i\}$, and this set is not necessarily consecutive. We first use their algorithm to compute the SCCs of the graph $\vec{G}\backslash s$. We can do this despite the fact that now a y node may be matched with more than one x node by merging its neighbors into one x node (note that the neighbors of this merged x node still form an interval of the y-nodes). Let C_s denote the SCC of the node s in $\vec{G}$. An SCC of G which is disjoint from C_s is also an SCC of $\vec{G}\backslash s$. C_s, however, contains the union of zero or more SCCs of $\vec{G}\backslash s$. We wish to find these SCCs and merge them to obtain C_s. Each of these SCCs has the property that it can reach s and can be reached from s. For each SCC C we compute two flags, *reached_from_s*$[C]$ and *reaches_s*$[C]$, and merge all components for which both flags are set to true into C_s.

We will use the following notation. For each SCC C of $\vec{G}\backslash s$, let *min_y*$[C] = \min\{i | y_i \in C\}$ and *max_y*$[C] = \max\{i | y_i \in C\}$ be the lowest and highest indices of y nodes in C. Moreover, let *reaches_left*$[C] = \min\{\underline{D}_j | x_j \in C\}$ and

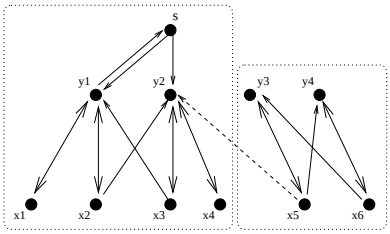

Fig. 4. The oriented intersection graph for our running example. It has two SCCs: $\{s, x_1, x_2, x_3, x_4, y_1, y_2\}$ and $\{x_5, x_6, y_3, y_4\}$. The dashed edge is inconsistent.

reaches_right$[C] = \max\{\overline{D}_j | x_j \in C\}$ be the minimum and maximum indices of y nodes that can be reached from C.

Lemma 8. *C can reach every y_i with reaches_left$[C] \leq i \leq$ reaches_right$[C]$ by one edge from an x node in C.*

Proof. The proof appears in [4]. It is simple and relies on the convexity. □

Now we will explain how to determine whether s can reach a component C. If this is the case then there is a path Q from s to some node y_i in C. And hence, if we delete the first edge of Q, we obtain a path P from a node $y_{i'}$ to y_i in $\vec{G}\backslash s$. In the lemma below, we state that this implies the existence of path P' from $y_{i'}$ to y_i that traverses the SCCs of $\vec{G}\backslash s$ in monotonous manner, either from left to right (cf. Condition (1)) or from right to left (cf. Condition (2)):

Lemma 9. *If there is a path from y_{i_1} to y_{i_k} in $\vec{G}\backslash s$ then there is a path $P = (y_{i_1}, x_{j_1}) \circ (x_{j_1}, y_{i_2}) \circ \cdots \circ (x_{j_{k-1}}, y_{i_k})$ in $\vec{G}\backslash s$ such that if we let C_κ denote the component of y_{i_κ} for each $1 \leq \kappa \leq k$, then one of the following holds:*

$$\forall_{1 \leq \kappa < k}\ min_y[C_\kappa] \leq min_y[C_{\kappa+1}] \quad \text{right-monotony} \tag{1}$$

$$\forall_{1 \leq \kappa < k}\ max_y[C_\kappa] \geq max_y[C_{\kappa+1}] \quad \text{left-monotony} \tag{2}$$

Proof. If P visits more than one component, let κ be the smallest index such that $C_\kappa \neq C_{\kappa+1}$. Assume that Condition (1) holds for κ. If P does not fulfil Condition (1), then there are three components C_1, C_2, C_3 which are visited consecutively by P in that order such that $min_y[C_1] < min_y[C_2]$ and $min_y[C_3] < min_y[C_2]$. We will show that there is a path which skips C_2 and goes directly from C_1 to C_3.
If $min_y[C_3] < min_y[C_1] < min_y[C_2]$ then by Lemma 8, C_2 can reach C_1 by a single edge, which closes a cycle that visits both of them. This means that $C_1 = C_2$, in contradiction to the assumption that they are distinct.
We therefore know that $min_y[C_1] < min_y[C_3] < min_y[C_2]$. Then by Lemma 8, C_1 can reach C_3 by one edge. So we can construct a path from y_{i_1} to y_{i_k} which visits the same components as P except that it goes from C_1 directly to C_3

without going through C_2. Since this decreases the number of transitions between components along the path, we can obtain a monotonous path by a finite number of such modifications.

Similarly, if Condition (2) holds for κ then we can find a path that fulfils Condition (2) for all $1 \leq \kappa < k$. Assume that neither condition holds. Then $min_y[C_{\kappa+1}] < min_y[C_\kappa]$ and $max_y[C_\kappa] < max_y[C_{\kappa+1}]$. By Lemma 8, $C_{\kappa+1}$ reaches back to C_κ, a contradiction. □

The algorithm in Figure 5 performs a left to right scan and marks all the components of $\vec{G}\backslash s$ that can be reached from s via a right-monotonous path. In the initialization phase it marks all components that can be reached from s by a single edge and during the scan it marks each component that can be reached from a scanned marked component by one edge. The running time is $O(n')$.

forall SCCs C of $\vec{G}\backslash s$ **do** *reached_from_s*$[C] \leftarrow false$
for $i = 1$ to n' **do if** (s, y_i) exists **then** *reached_from_s*$[C[y_i]] \leftarrow$ true
(* Scan from left to right and find the maximum y's reached from s ***)**
y_reached $\leftarrow 0$ **(*** Index of the rightmost y node that can be reached from s ***)**
for $i = 1$ to n' **do**
  $C \leftarrow C[y_i]$
  if *y_reached* $\geq i$ **then** *reached_from_s*$[C] \leftarrow$ true
  if *reached_from_s*$[C]$ **then** *y_reached* $\leftarrow \max\{$*y_reached*, *reaches_right*$[C]\}$
endfor

Fig. 5. Algorithm to mark SCCs that are reached from s via a right-monotonous path

Lemma 10. *The algorithm in Figure 5 sets reached_from_s*$[C]$ *to true iff s can reach C by a right-monotonous path.*

Proof. It is easy to see that if a component becomes marked, then it can be reached from s. For the converse, let $P = (s, y_{i_1}) \circ \cdots \circ (y_{i_{k-1}}, x_{j_{k-1}}) \circ (x_{j_{k-1}}, y_{i_k})$ be a path such that P without its first edge is right-monotonous. We will show by induction on the length of P that after the i_k-th iteration, the component C_k of y_{i_k} is marked "reached from s".
For $|P| = 1$: $P = (s, y_{i_1})$ and the component of y_{i_1} is marked before the scan.
For $|P| > 1$: By the induction hypothesis, *reached_from_s*$[C_{k-1}]$ is set to true after iteration *min_y*$[C_{k-1}]$.[3] Since $x_{j_{k-1}}$ is in C_{k-1}, *reaches_right*$[C_{k-1}] \geq i_k \geq$ *min_y*$[C_k]$. In iteration *min_y*$[C_{k-1}]$, *y_reached* is set to at least *min_y*$[C_k]$. By Lemma 9, we know that iteration *min_y*$[C_k]$ does not preceed iteration *min_y*$[C_{k-1}]$. Since the index *y_reached* can only increase, we get that in iteration *min_y*$[C_k]$, C_k will be marked as reached from s. □

[3] We may assume w.l.o.g. that if P visits a component C then it also visits the node $y_{min_y[C]}$.

A symmetric algorithm can be used to find the components that can be reached from s by a left-monotonous path. To find the components that can reach s, we can use the same approach: If we delete the last edge of a path in $\vec{G}$ from a node in C to s, we obtain a path in $\vec{G}\backslash s$. So we can mark all components that can reach s by two scans in time $O(n')$.

5 Narrowing the bounds of the x-ranges

Let S denote the set of all solutions of the constraint, as defined in the introduction, and for $j = 1, \ldots, n$ let S_j be the projection of S onto the jth component. We will discuss how to compute the values $\overline{S}_1, \ldots, \overline{S}_n$ in time $O(n + n')$, a symmetric procedure can be used for the lower endpoints of the narrowed ranges.

Consider a node x_j and let C be its SCC in $\vec{G}$. By Lemma 2, a value $i \in D_j$ is contained in S_j iff $y_i \in C$. Thus $\overline{S}_j$ is the index of the rightmost y node in C that is connected to x_j. Suppose the y nodes $y_{i_1}, \ldots, y_{i_k}$ in C are sorted such that $i_1 < \cdots < i_k$. Then $\overline{S}_j = i_\kappa$ where $i_\kappa \leq \overline{D}_j < i_{\kappa+1}$ (and $i_{k+1} = n' + 1$). So let us further assume that the x nodes $x_{j_1}, \ldots, x_{j_l}$ in C are sorted such that $\overline{D}_{j_1} \leq \cdots \leq \overline{D}_{j_l}$. Then we can determine $\overline{S}_j$ for every x_j in C by merging the sorted sequences $(i_1, \ldots, i_k)$ and $(\overline{D}_{j_1}, \ldots, \overline{D}_{j_l})$ in time $O(k + l)$.

We need to say how these sorted sequences are constructed. For the y's this can be done with bucketsort. We have a bucket for each component, and for $i = 1, \ldots, n'$, we append y_i to the bucket corresponding to its component. This takes time $O(n')$ and constructs the y-sequences for all SCCs of $\vec{G}$. Since we have a global sorting of the x's according to their upper range endpoints, we can use the same approach to sort them in time $O(n)$.

For the first SCC of our example we would merge the y sequence $(1, 2)$ with the x upper bounds $(2_{x_1}, 3_{x_2}, 3_{x_3})$ and narrow the domains of x_2 and x_3 from $[1, 3]$ to $[1, 2]$.

6 Narrowing the bounds of the count variables

This section deals with the projections of S onto its components $S_{n+1}, \cdots, S_{n+n'}$. We show how to compute lower and upper bounds for the values in S_{n+i} for each $i = 1, \cdots, n'$. For the rest of this section we consider a fixed graph G. We assume that the generalized matching algorithm has computed the α_i and β_i values for G and terminated without reporting failure. Now we show that in any generalized matching, the number of x nodes matched to each y_i are in a certain range that we can compute in linear time.

Lemma 11. *Let M be a generalized matching and for $i = 1, \cdots, n'$ let $\mu_i = |M(\{y_i\})|$. Then for all i*

$$\max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \mu_j\right) \leq \mu_i \leq \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \mu_j\right) \quad (*)$$

Proof. By the choice of the μ's we have $|M(\{y_1, \cdots, y_{i-1}\})| = n - \sum_{j=i}^{n'} \mu_j$. By Lemmas 4, 5 we know that $\alpha_{i-1} \leq |M(\{y_1, \cdots, y_{i-1}\})| \leq \beta_{i-1}$.
Therefore $\alpha_{i-1} \leq n - \sum_{j=i}^{n'} \mu_j \leq \beta_{i-1}$ or $n - \beta_{i-1} \leq \sum_{j=i}^{n'} \mu_j \leq n - \alpha_{i-1}$.
And hence $n - \beta_{i-1} - \sum_{j=i+1}^{n'} \mu_j \leq \mu_i \leq n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \mu_j$. □

This motivates the following definition. We call a sequence $(\mu_s, \cdots, \mu_{n'})$ a *legal count choice* iff inequality (*) is satisfied for all i in $[s, n']$. The Lemma above implies that any generalized matching M in G induces the legal choice $(|M(y_1)|, \ldots, |M(y_{n'})|)$. This allows us to compute for any y_i a lower bound l_i and an upper bound u_i on $|M(y_i)|$. Unfortunately there exist examples where our bounds are not tight.[4]

Apart from these bounds the algorithm below determines two count choices. The algorithm maintains the invariant that in iteration i both $(\kappa_i, \ldots, \kappa_{n'})$ and $(\lambda_i, \ldots, \lambda_{n'})$ are legal count choices such that for any legal count choice $(\mu_i, \ldots, \mu_{n'})$ we have $\sum_{j=i}^{n'} \kappa_j \leq \sum_{j=i}^{n'} \mu_j \leq \sum_{j=i}^{n'} \lambda_j$. As all sums that appear in the algorithm can be computed incrementally, the running time is $O(n')$.

for $i = n'$ to 1 **do**

$l_i \leftarrow \max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \lambda_j\right)$; $u_i \leftarrow \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \kappa_j\right)$

$\kappa_i \leftarrow \max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \kappa_j\right)$; $\lambda_i \leftarrow \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \lambda_j\right)$

endfor

We will now prove by induction on i that the algorithm computes the correct bounds for the count variables and that the invariant holds. For $i = n'$ Lemma 11 implies that $l_{n'}$ and $u_{n'}$ are lower and upper bounds. Since all sums are empty, we have $\kappa_{n'} = l_{n'}$ and $\lambda_{n'} = u_{n'}$, and hence, the claimed invariant holds.

For the induction step, we assume that our claim holds for $i+1$ and deduce it for i. By the left-hand side of inequality (*), we see that the minimum legal value for μ_i is obtained if $\sum_{j=i+1}^{n'} \mu_j$ is set to its largest possible value. Applying the invariant for $i+1$ this sum is maximized by the legal choice $(\lambda_{i+1}, \ldots, \lambda_{n'})$. So by Lemma 11, the algorithm computes a lower bound l_i. A similar argument holds for the upper bound u_i. By the definition of a legal choice, extending our two count choices by κ_i and λ_i respectively yields two legal choices again.

Fix a legal choice $(\mu_i, \ldots, \mu_{n'})$. What remains to prove are the two inequalities on the sum of the μ's.

$$\begin{aligned}
\textstyle\sum_{j=i}^{n'} \mu_j &= \mu_i + \textstyle\sum_{j=i+1}^{n'} \mu_j \overset{\text{L 11}}{\geq} \max(L_i, n - \beta_{i-1} - \textstyle\sum_{j=i+1}^{n'} \mu_j) + \textstyle\sum_{j=i+1}^{n'} \mu_j \\
&= \max(L_i + \textstyle\sum_{j=i+1}^{n'} \mu_j, n - \beta_{i-1}) \\
&\overset{\text{IH}}{\geq} \max(L_i + \textstyle\sum_{j=i+1}^{n'} \kappa_j, n - \beta_{i-1}) \\
&= \max(L_i, n - \beta_{i-1} - \textstyle\sum_{j=i+1}^{n'} \kappa_j) + \textstyle\sum_{j=i+1}^{n'} \kappa_j \\
&= \kappa_i + \textstyle\sum_{j=i+1}^{n'} \kappa_j = \textstyle\sum_{j=i}^{n'} \kappa_j
\end{aligned}$$

[4] Choosing $D_1 = [1, 3], D_2 = [2, 2]$ and $E_1 = E_2 = E_3 = [0, 1]$ is such an example. The lower endpoint of E_2 will not be narrowed to 1 by our algorithm.

An analogous computation shows $\sum_{j=i}^{n'} \mu_j \leq \sum_{j=i}^{n'} \lambda_j$, so we can derive:

Lemma 12. *Suppose that $S \neq \emptyset$ and let $\ell_1, \cdots, \ell_{n'}$ and $u_1, \cdots, u_{n'}$ be the values computed by our algorithm. Then $\underline{S}_{n+i} \geq \ell_i$ and $\overline{S}_{n+i} \leq u_i$ holds for $i = 1, \cdots, n$.*

On our running example, the algorithm computes the values listed below:

i	l_i	u_i	κ_i	λ_i	$\sum_{j=i}^{n'} \kappa_i$	$\sum_{j=i}^{n'} \lambda_i$
4	1	1	1	1	1	1
3	1	1	1	1	2	2
2	1	2	1	2	3	4
1	2	3	3	2	6	6

The c-ranges are therefore narrowed to $E_1 = [2, \mathbf{2}]$, $E_2 = [1, \mathbf{1}]$, $E_3 = [1, \mathbf{2}]$ and $E_4 = [0, \mathbf{1}]$, where the narrowed endpoints are typeset in bold.

7 Conclusion

We have designed a propagation algorithm for the Global Cardinality Constraint that achieves bound consistency for the assignment variables. We wish to point out that there are two possible implementations for the algorithm. One runs in time $O((n' + n) \log n)$, uses very simple data structures and performs well in practice. The other requires more elaborate data structures (for the offline-min computation) and achieves a running time of $O(n + n')$ plus the time required to sort the assignment variables by the endpoints of their ranges. In some cases the latter is asymptotically better.

In addition, we present an algorithm that narrows the bounds of the count variables, but it does not always achieve bounds consistency. So the following question remains open: Is there an efficient algorithm that achieves bound consistency for the count variables?

References

1. A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. H.N. Gabow and R.E. Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.
3. L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.
4. I. Katriel and S. Thiel. Fast bound consistency for the global cardinality constraint. Research Report MPI-I-2003-1-013, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2003.
5. K. Mehlhorn and S. Thiel. Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *LNCS*, pages 306–319, 2000.
6. J.-C. Régin. Generalized Arc-Consistency for Global Cardinality Constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 209–215, 1996.